Available online at www.sciencedirect.com

SCIENCE DIRECT®

Science of Computer Programming 55 (2005) 259–288

Science of Computer Programming

www.elsevier.com/locate/scico

Asynchronous system synthesis

J. Plosila[a], K. Sere[b,*], M. Waldén[b]

[a] *Department of Information Technology, University of Turku, Turku Centre for Computer Science (TUCS), FIN-20520 Turku, Finland*

[b] *Department of Computer Science, Åbo Akademi University, Turku Centre for Computer Science (TUCS), FIN-20520 Turku, Finland*

Received 31 August 2003; received in revised form 15 April 2004; accepted 30 May 2004
Available online 5 November 2004

Abstract

We propose a method for synthesising a set of components from a high-level specification of the intended behaviour of the target system. The designer proceeds via correctness-preserving transformation steps towards an implementable architecture of components which communicate asynchronously. The interface model of each component specifies the communication protocol used. At each step a pre-defined component is extracted and the correctness of the step is proved. This ensures the compatibility of the components. We use Action Systems as our formal approach to system design. The method is inspired by hardware-oriented approaches with their component libraries, but is more general. We also explore the possibility of using tool support to administer the derivation, as well as to assist in correctness proofs. Here we rely on the tools supporting the B Method, as this method is closely related to Action Systems and has good tool support.

1. Introduction

When carrying out formal *specification and derivation* of systems we can apply methods that allow a high-level abstract specification of a system to be stepwise developed into a more concrete version by correctness-preserving transformations. Hence, these methods

* Corresponding address: Department of Computer Science, Åbo Akademi University, Lemminkainengatan 14, SF-20520Åbo, Finland.
E-mail address: kaisa.sere@abo.fi (K. Sere).

0167-6423/$ - see front matter © 2004 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2004.05.018

provide a top-down approach to system design where the initial specification is a very abstract view of the system to be built. Details about the intended functionality and hardware/software components of the system are added stepwise to the specification during the design while preserving the intended observable behaviour of the original description. While this approach is very appealing from the designer's point of view as it allows the system to be developed and verified in manageable tasks, it still lacks e.g. good tool support.

In this paper, we concentrate on formal specification and derivation of *asynchronous systems* within the *Action Systems formalism*. Such systems may contain locally synchronous components, but the components interact with each other via asynchronous communication channels. This kind of architecture provides a promising, modular and reliable approach for implementing modern large digital systems. To handle the complexity in the design, component-based methods are needed and, hence, methods for guaranteeing component compatibility. We show how this is ensured within action systems.

Action Systems [3] and the associated *refinement calculus* [5], which provides a mathematical reasoning basis for the stepwise development of action systems, have shown their value in the design of reliable and correct systems in many ways [23]. The formal design methods supporting action systems and reasoning about them are heavily influenced by approaches to parallel and distributed program design as well as approaches to object-oriented programming [3,7]. Recently, methods for deriving systems from an abstract specification all the way down to VLSI circuits within Action Systems have received considerable attention [19,22]. An Action Systems-based design process of an asynchronous system starts from an abstract specification which is then decomposed into asynchronously communicating components. Each component can be implemented as an asynchronous (self-timed) or a synchronous (clocked) hardware module, or developed into an executable piece of software which is run in a standard microprocessor. Utilisation of formal methods is particularly important when designing complex embedded systems, as it potentially reduces the design costs by eliminating errors and wrong design decisions at an early stage of the design process, before the costly hardware modules involved, e.g. VLSI chips, have been manufactured.

In this paper we propose an approach to the synthesis of asynchronous networks of action system components from a high-level system specification using a pre-defined component library. This is a very important design phase as it determines the basic architectural structure of the target system. We assume that there is an implementation in the library for each component and here we concentrate on the specification of a component in terms of its interface model. As our formalism is state based, the interface model is given in terms of the interface variables. The model captures the types of the interface variables as well as the asynchronous communication protocols used to interact with the component. The protocols are given in terms of atomic actions operating on the state. The compatibility of components is shown within the refinement calculus during the synthesis process. The process itself proceeds by extracting components and decomposing the system. Compared to our earlier work in the same area [20] we present here a generic asynchronous component model and give more general transformation rules for component-based design.

In order to get more confidence in the development we like to have mechanical tool support. Atelier B [16] and B-Toolkit [17] provide this kind of support. They both comprise

a set of tools which support the B Method [1], a formal framework for stepwise system development. Action Systems and the B Method have essentially the same structure, both are state-based methods as opposed to event-based process calculi, and both support the stepwise refinement paradigm to system construction [26]. Recently, Waldén et al. [9,26] have shown how action systems can be modelled with the B Method. The B Method offers automatic proof support for the verification of the correctness of each step as well as tools for managing and administering a derivation task [17,16]. The designer supplies the tool with the specification and the refinements of this specification. The verification conditions, the proof obligations, needed for proving the correctness of the refinement steps can be automatically generated. Furthermore, these verification conditions can be automatically or interactively proved using these tools. Hence, Action Systems can utilise the mechanical tool support provided by the B Method. In this paper we extend the applicability area of the B Method to provide support for formal asynchronous system design by proposing two rather simple semantic constructs for the tool and their precise formal interpretation.

We start in Section 2 by introducing the Action Systems formalism embedded within the B Method. In Section 3 we formalise the concepts of asynchronous systems within the Action Systems approach and define our model for components. In Section 4 we develop the refinement and the component extraction ideas as well as the associated tool support needed in the design of asynchronous systems. Section 5 is devoted to the synthesis process. We show how asynchronously communicating components are brought into a system specification in a stepwise manner and show how component compatibility is ensured. Section 6 presents a case study on component-based design. We end in Section 7 with some concluding remarks.

2. Action systems in B

Systems are modelled in B via *abstract machines*. The main components of an abstract machine are the state variables, the operations on the state variables and the invariant giving the properties of these variables. For specifying the operations we use substitutions, for example, a *skip*-substitution, a simple substitution ($x := e$), a multiple substitution ($x := e \parallel y := f$), a sequential substitution ($x := e;\ y := f$), a preconditioned substitution (**PRE** P **THEN** S **END**), an *action* (also called guarded substitution) (**SELECT** P **THEN** S **END**) or a non-deterministic substitution (**ANY** x **WHERE** P **THEN** S **END**), where x and y are distinct variables, e and f are expressions, P is a predicate and S is a substitution.

Each substitution S is defined as a predicate transformer which transforms a post-condition Q into the weakest precondition for S to establish Q, $\mathrm{wp}(S, Q)$ [12], the initial states from which S is guaranteed to terminate. The substitutions above are defined as follows:

$$
\begin{array}{ll}
\mathrm{wp}(\mathit{skip}, Q) & = Q \\
\mathrm{wp}(x := e, Q) & = Q[x := e] \\
\mathrm{wp}(x := e \parallel y := f, Q) & = Q[x, y := e, f], \text{ where } x \cap y = \emptyset \\
\mathrm{wp}(x := e;\ y := f, Q) & = (Q[y := f])[x := e] \\
\mathrm{wp}(\textbf{PRE } P \textbf{ THEN } S \textbf{ END}, Q) & = P \wedge \mathrm{wp}(S, Q) \\
\mathrm{wp}(\textbf{SELECT } P \textbf{ THEN } S \textbf{ END}, Q) & = P \Rightarrow \mathrm{wp}(S, Q) \\
\mathrm{wp}(\textbf{ANY } x \textbf{ WHERE } P \textbf{ THEN } S \textbf{ END}, Q) & = (\forall x.P \Rightarrow \mathrm{wp}(S, Q))
\end{array}
$$

The abstract machine $\mathcal{A}$ given below,

MACHINE $\mathcal{A}$
VARIABLES
x
INVARIANT
$I(x)$
INITIALISATION
$x := x_0$
OPERATIONS
$A_1 \;\widehat{=}\;$ **SELECT** P_1 **THEN** S_1 **END**;
...
$A_m \;\widehat{=}\;$ **SELECT** P_m **THEN** S_m **END**
END

where every operation in the *operations*-clause is an action, is called an *action system*. An action system is identified by a unique name, here $\mathcal{A}$. The state variable(s) x of the action system are given in the *variables*-clause. The invariant $I(x)$ in the *invariant*-clause gives invariance properties for the variables. The variables are assigned initial values in the *initialisation*-clause. In the *operations*-clause each action A_i is given as a named operation. An action system can be composed in parallel with another action system (Section 3). In order to model communication between the action systems, each operation might have value parameters and/or return a result. An action A with the value parameter a and the result parameter b is denoted as $b \leftarrow A(a)$. The parameters are used to model communication in the form of message-passing.

Action systems are used as a model for parallel and distributed systems [3,26] with the basic idea that actions are selected for execution in a non-deterministic way. Hence, there is a non-deterministic choice between the actions $A_1, \ldots, A_m$ of $\mathcal{A}$, $A_1 \,[]\, \ldots \,[]\, A_m$. The non-deterministic choice of the actions A and B is defined as $\mathrm{wp}(A \,[]\, B, Q) = \mathrm{wp}(A, Q) \wedge \mathrm{wp}(B, Q)$. Only actions that are *enabled*, i.e., when the predicate P in the guarded substitution holds in a given state, are considered for execution. The behaviour of the action system is such that the initialisation statement is executed first. Thereafter, as long as there are enabled actions, one enabled action at a time is chosen and executed as an *atomic* entity. The execution terminates when there are no enabled actions. If two actions do not share any variables, they can be executed in any order or in parallel. Hence, we have an interleaving semantics for action systems.

Example. The machine $\mathcal{E}$ below is a high-level action system specification of a system unit which computes a new value for data *dout* whenever the action E is enabled and selected for execution. The parameters l and r act as two-directional communication signals of $\mathcal{E}$. In other words, when $\mathcal{E}$ receives values for l and r from another action system, it responds by assigning other values which can then be detected by this action system. The values l and r that can be assigned are *req* and *ack* corresponding to the request and acknowledgement phases of asynchronous communication (Section 3.1). The machine $\mathcal{E}$ receives input data *din* when $l = req$ and sends output data *dout* by setting r to *req*:

MACHINE $\mathcal{E}$
OPERATIONS
$l, r, dout \leftarrow E(l, din, r, dout) \;\hat{=}$
SELECT $l = req \wedge r = ack$
THEN ANY $dout'$ **WHERE** $F(dout', din, dout)$ **THEN** $dout := dout'$ **END**
$\| r := req \| l := ack$
END
END

2.1. Scheduling of actions

Implicitly there is a non-deterministic choice between enabled actions in the *operations*-clause as explained above. Sometimes we need to express this policy explicitly in a *scheduling*-clause of an action system. In this clause we can also give other more specific scheduling policies like sequential composition or some parallel, exclusive, prioritised, or probabilistic composition between actions. We consider the scheduling to be an iterative construction modelling the iterative behaviour of action systems. The clause is optional, but in the case where a *scheduling*-clause appears in an action system, all the actions of the machine must be included. The scheduling has the same name as the abstract machine. If the actions have parameters, they will also be parameters of the scheduling.

MACHINE $\mathcal{A}$
...
SCHEDULING
$\mathcal{A} \;\hat{=}\; schedule_{\mathcal{A}}(A_1, \ldots, A_m)$
END

where $schedule_{\mathcal{A}}(A_1, \ldots, A_m)$ is of the following form:

$$\begin{array}{lll} schedule_{\mathcal{A}}(A_1, \ldots, A_m) & = & \text{"("}\ P\ \text{")"} \\ P & = & \text{"("}\ P \bullet P\ \text{")"} \mid P_A \\ \bullet & = & \ ; \mid [] \end{array}$$

where $P_A \in \{A_1, \ldots, A_m\}$. Each action of $\mathcal{A}$ should appear once and only once in the *scheduling*-clause. Parentheses are used to structure the scheduling. In this paper we focus on the two most common operators of composition between actions: non-deterministic choice, [] ; and (non-atomic) sequential composition, **;**. The non-atomic sequential composition is frequently needed in asynchronous modelling to sequence communication events on asynchronous *communication channels* discussed later in Section 3.1.

As an example of the scheduling, let the action system $\mathcal{A}$ have three actions A_1, A_2 and A_3 and the *scheduling*-clause $\mathcal{A} \;\hat{=}\; ((A_1 \;;\; A_2) \;[]\; A_3)$. The execution of $\mathcal{A}$ is restricted so that A_1 and A_2 always are executed in a sequence interleaved with A_3. Hence, the execution order of $\mathcal{A}$ can be: (A_1, A_3, A_2), if action A_3 is enabled after A_1 has been executed. We remind the reader of the iterative behaviour of action systems. In the case where the actions have parameters these have to be taken into account in the scheduling to show the communication. For example, the sequential execution of the two actions

$B_1(b)$ and $d \leftarrow B_2(c)$ with associated value and result parameters b, c and d is given as $\mathcal{B}(b, c, d) \mathrel{\hat{=}} B_1$ **;** B_2 in the *scheduling*-clause of $\mathcal{B}$.

The *scheduling*-clause is not part of B, but is introduced here as a derived construct. The non-deterministic choice was explained above. The sequential composition of two actions can also be interpreted in terms of the non-deterministic choice. Let us consider the two actions $A \mathrel{\hat{=}}$ **SELECT** P **THEN** S **END** and $B \mathrel{\hat{=}}$ **SELECT** Q **THEN** T **END**. Their sequential composition, A **;** B, can then be interpreted as the non-deterministic choice, A^{pc} [] B^{pc}, between the two actions A^{pc} and B^{pc}, where a variable pc (initially set to 1) for scheduling the actions has been added:

$$A^{pc} \mathrel{\hat{=}} \textbf{SELECT } P \wedge pc = 1 \textbf{ THEN } S \parallel pc := 2 \textbf{ END}$$
$$B^{pc} \mathrel{\hat{=}} \textbf{SELECT } Q \wedge pc = 2 \textbf{ THEN } T \parallel pc := 1 \textbf{ END}$$

Hence, B^{pc} is only enabled after A^{pc} has been executed, setting pc to 2. We can note that the *scheduling*-clause provides us with a convenient way of rewriting the scheduling of the actions, which otherwise should be coded within the actions.

Example. In the machine $\mathcal{R}eg$ below the operations Reg_1 and Reg_2 are sequentially composed. We return to this machine later.

MACHINE $\mathcal{R}eg$
OPERATIONS
$b, dout \leftarrow Reg_1(a, din) \mathrel{\hat{=}}$ **SELECT** $a = req$ **THEN** $dout := din \parallel b := req$ **END**;

$a \leftarrow Reg_2(b) \mathrel{\hat{=}}$ **SELECT** $b = ack$ **THEN** $a := ack$ **END**
SCHEDULING
$\mathcal{R}eg(a, din, b, dout) = Reg_1$ **;** Reg_2
END

3. Modularisation

Asynchronous interfacing provides a viable approach for building modern large digital systems, composed of several hardware and software units, in a modular and reliable manner. In asynchronous communication, an event of data transfer between two system components consists of two phases: request and acknowledgement (Section 3.1). Depending on the application, the duration of each phase may be either unbounded or bounded. Asynchronously communicating components form an asynchronous system architecture in which a component module, taken separately, can internally be either an asynchronous (self-timed) or synchronous (clocked) hardware block, or a software module running in a standard or application-specific processor.

Action systems can be composed/decomposed into parallel systems [4]. The parallel composition of action systems $\mathcal{A}$ and $\mathcal{B}$ can be presented in the B Method using the *extends*-clause. This also provides an efficient way to model system hierarchy, i.e., a system can be modelled as a composition of subsystem modules listed in its *extends*-clause.

MACHINE $\mathcal{A}$
EXTENDS $\mathcal{B}$
VARIABLES
x
INVARIANT
$I(x)$
INITIALISATION
$x := x_0$
OPERATIONS
$A(a)$ $\hat{=}$ **SELECT** P **THEN** S **END**
SCHEDULING
$\mathcal{A}(a) \hat{=} A \,[]\, \mathcal{B}(a, x)$
END

MACHINE $\mathcal{B}$
VARIABLES
y
INVARIANT
$J(y)$
INITIALISATION
$y := y_0$
OPERATIONS
$B(b, c)$ $\hat{=}$ **SELECT** Q **THEN** T **END**
SCHEDULING
$\mathcal{B}(b, c) \hat{=} B$
END

Here the action system $\mathcal{A}$ *extends* the system $\mathcal{B}$ indicating that $\mathcal{A}$ is considered to be composed in parallel with $\mathcal{B}$. We can also say that $\mathcal{A}$ contains the component (i.e. subsystem) module $\mathcal{B}$. The scheduling of $\mathcal{A}$ is then $A \,[]\, \mathcal{B}(a, x)$, where a and x are the actual parameters which should be variables of $\mathcal{A}$ and/or formal parameters of the actions in $\mathcal{A}$. The variable(s) y in $\mathcal{B}$ should be distinct from the variables x in $\mathcal{A}$, $(x \cap y = \emptyset)$. The result of composing $\mathcal{A}$ and $\mathcal{B}$ in parallel is given as the system $\mathcal{AB}$ below.

MACHINE $\mathcal{AB}$
VARIABLES
x, y
INVARIANT
$I(x) \wedge J(y)$
INITIALISATION
$x := x_0 \,||\, y := y_0$
OPERATIONS
$A(a)$ $\hat{=}$ **SELECT** P **THEN** S **END**
$B(a)$ $\hat{=}$ **SELECT** $Q[a/b, x/c]$ **THEN** $T[a/b, x/c]$ **END**
SCHEDULING
$\mathcal{AB}(a) \hat{=} A \,[]\, B$
END

The variables, the invariants and the actions of the two action systems $\mathcal{A}$ and $\mathcal{B}$ are simply merged in the composed action system $\mathcal{AB}$. The formal parameters, b and c, in the action of $\mathcal{B}$ are replaced by the actual parameters, a and x, in $\mathcal{A}$. Since a is a formal parameter of the scheduling $\mathcal{A}$, it should also be a formal parameter of the action B after the substitution.

Example. To exemplify modularisation, let us consider the action system $\mathcal{E}^1$ below which contains the system $\mathcal{R}eg$ (described in Section 2.1) as a component. The scheduling of $\mathcal{E}^1$ is $((E_{11} \,;\, E_{13}) \,[]\, E_{12}) \,[]\, \mathcal{R}eg(c_1, dm, c_2, dout)$, where $\mathcal{R}eg(c_1, dm, c_2, dout)$ stands for $(Reg_1 \,;\, Reg_2)$, indicating that the action systems $\mathcal{E}^1$ and $\mathcal{R}eg$ are composed in parallel. The types of the variables involved are given as the sets $com = \{req, ack\}$ and $data$ (any data type). We will return to this machine later.

MACHINE $\mathcal{E}^1$
EXTENDS
$\mathcal{R}eg$
VARIABLES
c_1, c_2, dm
INVARIANT
$c_1 \in com \wedge c_2 \in com \wedge dm \in data$
INITIALISATION
$c_1 := ack \;||\; c_2 := ack \;||\; dm :\in data$
OPERATIONS
$E_{11}(l, din, dout) \;\hat{=}$
SELECT $l = req \wedge r = ack$
THEN **ANY** dm' **WHERE** $F(dm', din, dout)$ **THEN** $dm := dm'$ **END**
$|| \; c_1 := req$
END;

$r \leftarrow E_{12} \;\hat{=}$ **SELECT** $c_2 = req$ **THEN** $c_2 := ack \;||\; r := req$ **END**;

$l \leftarrow E_{13} \;\hat{=}$ **SELECT** $c_1 = ack$ **THEN** $l := ack$ **END**
SCHEDULING
$\mathcal{E}^1(l, din, r, dout) \;\hat{=}\; (((E_{11} \;;\; E_{13}) \;[]\; E_{12}) \;[]\; \mathcal{R}eg(c_1, dm, c_2, dout))$
END

3.1. Asynchronous communication channels

In this paper, we focus on modelling and derivation of systems which are organisations of asynchronously communicating components. Such building blocks with asynchronous interfaces are here collectively called *asynchronous components*, independently of the intended internal structure of each component. As an example, the abstract system $\mathcal{E}$ discussed in Section 2 acts as an asynchronous component towards its environment.

Interaction between asynchronous components is arranged via *communication channels* composed of the *value and result parameters* of the actions. In our formal design framework, a communication channel $c(d)$, or a channel c for short, is defined to be a tuple (c, d), where c is a *communication variable* and d the list of those *data variables* whose values are transferred from a system module to another by communicating via c. Hence, the variables c and d are *interface variables* of an asynchronously communicating component. Logical variables that are not directly concerned with the communication can, however, also be interface variables. In the case when the list d is empty, c is called a *control channel*. Otherwise we have a *data channel* $c(d)$. Furthermore, when referring to a single party of communication, we talk about *communication ports* rather than channels.

Generally, a communication variable c is of the enumerated type $com_{m,n}$ defined by

$$com_{m;n} \mathrel{\hat{=}} \{req_1, \ldots, req_m, ack_1, \ldots, ack_n\}$$

where $req_1, \ldots, req_m$ and $ack_1, \ldots, ack_n$ are request and acknowledgement states (values), respectively. A variable $c \in com_{m,n}$ is typically initialised to one of its acknowledgement states ack_j. If $m = 1$ or $n = 1$, the default value is just req or ack, respectively. Hence, the simplest and the most usual type $com_{1,1}$ is equivalent to $\{req, ack\}$ by default. We denote $com_{1,1}$ simply by com.

A communication channel connects two action system components, one of which acts as the *master* and the other as the *slave*. The master side of a channel is called an *active* communication port, and the slave side is referred to as a *passive* communication port. A *communication cycle* on a channel $c(d)$ includes two main phases. When the active party, the master, initiates the cycle by setting c to a request state req_i, the cycle is said to be in the *request phase*. Correspondingly, when the passive party, the slave, responds by setting c to an acknowledgement state ack_j, the communication cycle on c is said to be in the *acknowledgement phase*. The data d can be transferred either in the request phase from the master to the slave (*push channel*), in the acknowledgement phase from the slave to the master (*pull channel*), or in both phases bidirectionally (*biput channel*) [18].

Example. The above machine $\mathcal{E}^1$ and its component $\mathcal{R}eg$ (Section 2.1) communicate asynchronously via the push channels $c_1(dm) \in com$ and $c_2(dout) \in com$. When the system $\mathcal{E}^1$ transfers data *dm* to $\mathcal{R}eg$ by setting c_1 to the request state *req*, it acts as the master towards the machine $\mathcal{R}eg$ which sets c_1 to the acknowledgement state *ack* as a response. On the other hand, when data *dout* is transferred from $\mathcal{R}eg$ to $\mathcal{E}^1$ via the channel c_2, the system $\mathcal{R}eg$ is the active party and $\mathcal{E}^1$ acts as the slave.

3.2. Component model

Generally, an asynchronous component $\mathcal{M}$ has k active communication ports $a_i(da_i) \in com_{ni,mi}$, with $1 \leq i \leq k$, and l passive communication ports $p_j(dp_j) \in com_{nj,mj}$, with $1 \leq j \leq l$. Here da_i and dp_j represent the corresponding, possibly empty, lists of data parameters, and $k, l \geq 0$. A sketch of such a component $\mathcal{M}$ is shown below. In this abstract model, the operations A_i and P_j represent the actions dealing with communication on the active ports $a_i(da_i)$ and the passive ports $p_j(dp_j)$, respectively. Observe, however, that for some values of i and j the identifiers A_i and P_j may actually refer to the same action, because a single action can take care of communication events on several different ports. The values *req* and *ack* assigned to the ports represent any request and acknowledgement states of the ports of an actual component. The identifiers SA_i and SP_j, in turn, are arbitrary substitutions accessing the parameters da_i and dp_j, respectively. Possible local operations of $\mathcal{M}$, i.e., actions that access possible local variables only, are not shown below.

MACHINE $\mathcal{M}$
...
OPERATIONS
$a_1, da_1 \leftarrow A_1(a_1, da_1) \;\hat{=}$
  SELECT $a_1 = ack$ **THEN** $SA_1(da_1) \,||\, a_1 := req$ **END**;
$a_2, da_2 \leftarrow A_{2,1}(da_2) \;\hat{=}$
  SELECT $true$ **THEN** $SA_{2,1}(da_2) \,||\, a_2 := req$ **END**;
$da_2 \leftarrow A_{2,2}(a_2, da_2) \;\hat{=}$
  SELECT $a_2 = ack$ **THEN** $SA_{2,2}(da_2)$ **END**;
...
$p_1, dp_1 \leftarrow P_1(p_1, dp_1) \;\hat{=}$
  SELECT $p_1 = req$ **THEN** $SP_1(dp_1) \,||\, p_1 := ack$ **END**;
$dp_2 \leftarrow P_{2,1}(p_2, da_2) \;\hat{=}$
  SELECT $p_2 = req$ **THEN** $SP_{2,1}(dp_2)$ **END**;

$p_2, dp_2 \leftarrow P_{2,2}(dp_2) \;\hat{=}$
SELECT $true$ **THEN** $SP_{2,2}(dp_2) \;||\; p_2 := ack$ **END**;
...
SCHEDULING
$\mathcal{M}(a_1, da_1, a_2, da_2, \ldots, p_1, dp_1, p_2, dp_2, \ldots) \;\hat{=}$
$A_1 \;[]\; (A_{2,1} \;;\; L_A \;;\; A_{2,2}) \;[]\; \ldots \;[]\; P_1 \;[]\; (P_{2,1} \;;\; L_P \;;\; P_{2,2}) \;[]\; \ldots$
END

Notice that only the ports with $i, j \in \{1, 2\}$ are explicitly considered to demonstrate how a communication cycle on a port may involve either one operation (A_1 or P_1) or two operations ($A_{2,1}$ and $A_{2,2}$, or $P_{2,1}$ and $P_{2,2}$). In the latter case, the actions are often sequentially scheduled as shown in the above model $\mathcal{M}$. The symbol L_A (L_P) in the *scheduling*-clause represents all those actions which are sequentially scheduled between $A_{2,1}$ and $A_{2,2}$ ($P_{2,1}$ and $P_{2,2}$) and do not operate on the communication port $a_2(da_2)$ ($p_2(dp_2)$).

The properties of the local variables of the component are given in the invariant. The interface variables are, though, specified in the precondition of each action. For example, the action A_1 with the parameters a_1 modelling a channel and da_1 modelling data is given as

$a_1, da_1 \leftarrow A_1(a_1, da_1) \;\hat{=}$
PRE $a_1 \in com \wedge da_1 \in data$ **THEN**
SELECT $a_1 = ack$ **THEN** $SA_1(da_1) \;||\; a_1 := req$ **END**
END

For readability of the actions we leave out the preconditions with the type declarations of the internal variables in this paper.

Example. The machine $\mathcal{E}$ (Section 2) has the passive input port $l(din) \in com$, and the active output port $r(dout) \in com$. The single operation E of $\mathcal{E}$ takes care of communication on both ports. Consequently, it has the role of the actions A_1 and P_1 of the above abstract model $\mathcal{M}$. Also the component $\mathcal{R}eg$ (Section 2.1) has two communication ports: the passive input port $a(din) \in com$ and the active output port $b(dout) \in com$. The two sequentially scheduled operations Reg_1 and Reg_2 take care of communication on both ports, and hence they correspond to the operations $A_{2,1}$ and $A_{2,2}$ as well as $P_{2,1}$ and $P_{2,2}$ of the abstract component model $\mathcal{M}$.

4. Abstract machine refinement

Refinement is a viable method for stepwise derivation of systems. Let us consider an abstract machine $\mathcal{A}$ and its refinement $\mathcal{C}$ given below. The machine refinement states in the *refines*-clause what it refines, an abstract machine or another machine refinement. Below, the refinement $\mathcal{C}$ refines the abstract machine $\mathcal{A}$. The invariant $R(x, y)$ of the refinement gives the relation between the variable(s) x in the action system $\mathcal{A}$ and the variable(s) y in its refinement $\mathcal{C}$ for replacing abstract statements with more concrete ones. The refined and more concrete actions C_i are given in the *operations*-clause and the *scheduling*-clause indicates how these actions are composed.

MACHINE $\mathcal{A}$
VARIABLES
x
INVARIANT
$I(x)$
INITIALISATION
$x := x_0$
OPERATIONS
$A_1 \;\hat{=}\;$ **SELECT** P_1 **THEN** S_1 **END**;
...
$A_m \;\hat{=}\;$ **SELECT** P_m **THEN** S_m **END**
SCHEDULING
$\mathcal{A} \hat{=} \mathit{schedule}_{\mathcal{A}}(A_1, \ldots, A_m)$
END

REFINEMENT $\mathcal{C}$
REFINES
$\mathcal{A}$
VARIABLES
y
INVARIANT
$R(x, y)$
INITIALISATION
$y := y_0$
OPERATIONS
$C_1 \;\hat{=}\;$ **SELECT** Q_1 **THEN** T_1 **END**;
...
$C_n \;\hat{=}\;$ **SELECT** Q_n **THEN** T_n **END**
SCHEDULING
$\mathcal{C} \hat{=} \mathit{schedule}_{\mathcal{C}}(C_1, \ldots, C_n)$
MAPPINGS
...
END

We introduce a *mappings*-clause to explicitly state the refinement relation between the actions in $\mathcal{A}$ and $\mathcal{C}$. We rely here on data refinement, where we might for instance decrease the level of non-determinism of the statement or merely change the data representation. In the case where each action A_i in $\mathcal{A}$ is data refined by one action C_i in $\mathcal{C}$ $(n = m)$, we have an entry $A_i \leq C_i$ in the *mappings*-clause for each action A_i of $\mathcal{A}$ indicating that action A_i is data refined by C_i under invariant R.

MAPPINGS
$A_1 \leq C_1$,
...
$A_m \leq C_m$

In this case the scheduling of the corresponding actions should be the same in $\mathcal{A}$ and $\mathcal{C}$, i.e. if $\mathcal{A} \hat{=} A_1 \,[]\, A_2$ then $\mathcal{C} \hat{=} C_1 \,[]\, C_2$.

The B Method supports this one-to-one refinement. This is, however, too restrictive for derivation of asynchronous systems. In asynchronous system design we often rely on atomicity refinement [15,2,24], where we split an atomic action in a system into several actions in order to increase the degree of parallelism in the system. Hence, we may need to introduce new actions during the refinement process. In the case where we introduce new actions in $\mathcal{C}$, $(n > m)$, we have the case that an action in $\mathcal{A}$ is refined by a composition of actions in $\mathcal{C}$, e.g., $A_2 \leq C_3 \,[]\, C_4$. By introducing the *mappings*-clause, we may have tool support for this flexibility in the development. Each action of $\mathcal{A}$ and $\mathcal{C}$ should appear once and only once in this clause.

MAPPINGS
$A_i \leq (C_j \,[]\, \ldots \,[]\, C_l)^*$,
...

where $1 \leq i \leq m$ and $1 \leq j < l \leq n$. The entry states that the action A_i in $\mathcal{A}$ is data

refined by the iterative composition of the actions $C_j, \ldots, C_l$ in $\mathcal{C}$. In the *mappings*-clause we give a non-deterministic choice between the actions of $\mathcal{C}$ and denote iteration with a star *, indicating that all the actions $C_j, \ldots, C_l$ together refine A_i.

Notice that in the case where the refined action system $\mathcal{C}$ is composed in parallel with another action system $\mathcal{B}$, i.e., $\mathcal{C}$ has $\mathcal{B}$ as a component module, and $\mathcal{B}$ contains the scheduling $\mathcal{B} \;\hat{=}\; B_1 \,;\, B_2$, then either $\mathcal{B}$, or both B_1 and B_2 should appear within the composed actions in the *mappings*-clause of $\mathcal{C}$. This is due to the fact that $\mathcal{C}$ *extends* $\mathcal{B}$ and the actions of $\mathcal{B}$ are by definition considered to be actions in $\mathcal{C}$, as well as the fact that each action of $\mathcal{C}$ should appear once in the *mappings*-clause. For instance, if the action A_1 in $\mathcal{A}$ is refined by $(C_1 \,[]\, \mathcal{B})^*$ in $\mathcal{C}$ then we actually consider A_1 to be refined by the composition of C_1 and all the actions in $\mathcal{B}$, $A_1 \leq (C_1 \,[]\, B_1 \,[]\, B_2)^*$. In the *mappings*-clause of $\mathcal{C}$, a single action B_i of the component module $\mathcal{B}$ can be addressed using the dot notation $\mathcal{B}.B_i$. Hence, the mapping entry $A_1 \leq (C_1 \,[]\, \mathcal{B})^*$ can be written as $A_1 \leq (C_1 \,[]\, \mathcal{B}.B_1 \,[]\, \mathcal{B}.B_2)^*$.

In this paper we propose to merely add new variables in each refinement step. This gives us the case where the parallel composition of action systems is monotonic with respect to this refinement [6]. Due to this and to the transitivity of the refinement relation, if action system $\mathcal{A}$ is refined by $\mathcal{A}'$ and action system $\mathcal{B}$ is refined by $\mathcal{B}'$, then the parallel composition of $\mathcal{A}$ and $\mathcal{B}$ is refined by the parallel composition of $\mathcal{A}'$ and $\mathcal{B}'$. This means that the subsystems in a parallel composition may be refined independently.

In order to prove that the refinement $\mathcal{C}$ is a refinement of the action system $\mathcal{A}$ using the invariant R, a number of proof obligations must be satisfied [1]. These proof obligations are described in Appendix A. The *scheduling*- and *mappings*-clauses enable an automatic generation and check of these proof obligations. With the help of Event B [10], a version of Atelier B that is intended for developing distributed systems, we can generate the necessary proof obligations. In Event B several operations can refine the same operation and one operation can in turn refine several operations. However, a composition of operations cannot directly refine an operation in Event B.

5. Synthesis process

Top-down design of an asynchronous system starts from a high-level action system specification of the basic functionality of the target system. The initial abstract specification is then stepwise implemented, within the refinement calculus framework, as an asynchronous architecture of action system modules representing pre-defined library components available to the designer. Such a module could be, for example, an arithmetic logical unit, a computational algorithm, an interface module, a memory block, a controller of a set of other units, or even a very complex microprocessor core. After this *synthesis* process, the main system does not contain any operations of its own, but all of them come from the component modules listed in the *extends*-clause of the system. The components interact via asynchronous communication channels created during the stepwise extraction process.

The idea in the stepwise synthesis process of an asynchronous system is to refine the specification, by systematically modifying actions and introducing new asynchronous

communication channels, into a network of components existing in some module libraries. This is based on creating models of pre-defined library components into the system, and then replacing these embedded models with references to the actual library components. Each synthesis step creates a new such component reference and consists of a number of disciplined refinements, depending on the complexity of the library module involved. We give the refinement here as abstract templates that can be specialised for every particular case. The refinements are also presented so that many of the otherwise necessary proof obligations are satisfied here by construction and hence need not be proved separately. These transformations include introduction of one or more communication channels and decomposition of atomic actions into several separate actions using the introduced channels and sequential scheduling discussed in Section 2.1. The structure of asynchronous components was generally addressed in Section 3.2.

The major motivation for such component-oriented system refinement is that the library modules have pre-defined and pre-verified hardware or software implementations. This means that the designer does not have to further refine the components, which makes the design process more straightforward and efficient. Naturally, the method of implementing a system with pre-defined components can also be used for creating or extracting new library components which are to be explicitly refined into the implementation level and then re-used in future design projects as pre-defined components.

5.1. Synthesis step

Let us now consider a synthesis step from a more formal perspective. First, assume that we have a module library at our disposal, and that this library contains a component $\mathcal{M}$ with which we want to implement a part of the target system. Let $\mathcal{M}$ be an n-port module of the form

MACHINE $\mathcal{M}$
...
VARIABLES l
...
OPERATIONS
$m_1, dm_1 \leftarrow M_1(m_1, dm_1) \;\hat{=}\; \ldots$;
...
$m_n, dm_n \leftarrow M_n(m_n, dm_n) \;\hat{=}\; \ldots$;
$L \;\hat{=}\; \ldots$
SCHEDULING
$\mathcal{M}(m_1, dm_1, \ldots, m_n, dm_n) \;\hat{=}\; schedule_{\mathcal{M}}(M_1, \ldots, M_n, L)$
END

where $m_i(dm_i)$, $1 \leq i \leq n$, are the communication ports of $\mathcal{M}$, M_i represent all operations involved in the communication cycles on $m_i(dm_i)$ (cf. Section 3.2), L represents all operations accessing only local variables l of $\mathcal{M}$, and $schedule_{\mathcal{M}}$ denotes the scheduling of the specified operations in $\mathcal{M}$.

Assume that the specification of the system $\mathcal{A}$ we are developing is of the form

MACHINE $\mathcal{A}$
...
OPERATIONS
$A \mathrel{\hat{=}} \ldots ;$
...
SCHEDULING
$\mathcal{A}(\ldots) \mathrel{\hat{=}} schedule_{\mathcal{A}}(A, \ldots)$
END

where A symbolises all operations that are involved in the synthesis step at hand, i.e., those operations of $\mathcal{A}$ which we plan to implement with the above library component $\mathcal{M}$. This extraction procedure consists of a number of steps, or refinements, including introduction of the variables c_i and d_i, $1 \leq i \leq n$, which are to be assigned to the communication port parameters m_i and dm_i of $\mathcal{M}$, respectively. The goal is to gradually decompose the operations A of $\mathcal{A}$ into the operations $M_{\mathcal{A},i}$ which access the new variables c_i and d_i and which mimic the actions M_i of $\mathcal{M}$. Furthermore, if $\mathcal{M}$ contains some actions L which access some local variables l only, we also introduce the corresponding local variables $l_{\mathcal{A}}$ into $\mathcal{A}$ and extract the corresponding operations $L_{\mathcal{A}}$, in addition to the operations $M_{\mathcal{A},i}$, from the actions A of $\mathcal{A}$. These transformations are formalised below as a refinement rule.

With the added operations and variables the resulting system $\mathcal{A}'$, which is a refinement of the original system $\mathcal{A}$, has the form

REFINEMENT $\mathcal{A}'$
REFINES $\mathcal{A}$
...
VARIABLES $c_1, d_1, \ldots, c_n, d_n, l_{\mathcal{A}}$
...
OPERATIONS
$M_{\mathcal{A},1} \mathrel{\hat{=}} \ldots ;$
...
$M_{\mathcal{A},n} \mathrel{\hat{=}} \ldots ;$
$L_{\mathcal{A}} \mathrel{\hat{=}} \ldots ;$
$A' \mathrel{\hat{=}} \ldots ;$
...
SCHEDULING
$\mathcal{A}'(\ldots) \mathrel{\hat{=}} schedule_{\mathcal{A}'}(M_{\mathcal{A},1}, \ldots, M_{\mathcal{A},n}, L_{\mathcal{A}}, A', \ldots)$
MAPPINGS
$A \leq (A' \,[]\, M_{\mathcal{A},1} \,[]\, \ldots \,[]\, M_{\mathcal{A},n} \,[]\, L_{\mathcal{A}})^*$
...
END

where A' represents actions that are left of the initial actions A after the decomposition. The idea is that the refinement process is carried out in such a way that an embedded instance of the library component $\mathcal{M}$ is step by step created into the system. This means that the scheduling of the operations in $\mathcal{A}'$ has to be such that

$$schedule_{\mathcal{A}'} \mathrel{\widehat{=}} schedule_{\mathcal{M}}[M_{\mathcal{A},1}, \ldots, M_{\mathcal{A},n}, L_{\mathcal{A}} \,/\, M_1, \ldots, M_n, L] \,[]\, schedule_{\mathcal{A}''}(A', \ldots).$$

Then we can complete the synthesis step by replacing the local variables $l_{\mathcal{A}}$, the operations $M_{\mathcal{A},i}$ and $L_{\mathcal{A}}$, and the scheduling of these actions with a reference to the library component $\mathcal{M}$ mentioning it in the *extends*-clause of the system. This yields the machine $\mathcal{A}''$ given below. In order to show the correctness of this final transformation as a refinement step (and hence, the compatibility of the component) we need to verify the proof obligations (1)–(4) from Appendix A. They can be automatically generated and proved with the tools supporting the B Method.

REFINEMENT $\mathcal{A}''$
REFINES $\mathcal{A}'$
EXTENDS $\mathcal{M}$
...
VARIABLES $c_1, d_1, \ldots, c_n, d_n$
...
OPERATIONS
$A'' \;\hat{=}\; \ldots\;;$
...
SCHEDULING
$\mathcal{A}''(\ldots) \hat{=} \mathit{schedule}_{\mathcal{A}''}(A'', \ldots) \,[]\, \mathcal{M}(c_1, d_1, \ldots, c_n, d_n)$
MAPPINGS
$A' \leq A''$,
$M_{\mathcal{A},1} \leq \mathcal{M}(c_1, d_1, \ldots, c_n, d_n).M_1$,
...
$M_{\mathcal{A},n} \leq \mathcal{M}(c_1, d_1, \ldots, c_n, d_n).M_n$,
$L_{\mathcal{A}} \;\leq\; \mathcal{M}(c_1, d_1, \ldots, c_n, d_n).L[l_{\mathcal{A}}/l]$,
...
END

Example. Consider again the system $\mathcal{E}^1$ (Section 3) and its component $\mathcal{R}eg$ (Section 2.1). This composition has been derived from the initial machine $\mathcal{E}$ (Section 2) by splitting the operation E, which corresponds to the action A in the above discussion, into five separate parts by introducing the new variables $c_1, c_2 \in \mathit{com}$ and $\mathit{dm} \in \mathit{data}$. The operations E_{11}, E_{12} and E_{13} of $\mathcal{E}^1$ together correspond to the actions A' of the above system $\mathcal{A}''$, and the operations Reg_1 and Reg_2 of $\mathcal{R}eg$ correspond to the actions M_i $(1 \leq i \leq n)$ of the above abstract library component $\mathcal{M}$. Hence, the intermediate system, say $\mathcal{E}'$, which corresponds to the above machine $\mathcal{A}'$, is composed of the above-mentioned operations E_{11}, E_{12} and E_{13}, as well as the component-specific operations $\mathit{Reg}_{\mathcal{E},1}$ and $\mathit{Reg}_{\mathcal{E},2}$ defined as

$\mathit{Reg}_{\mathcal{E},1} \;\hat{=}\;$ **SELECT** $c_1 = \mathit{req}$ **THEN** $\mathit{dout} := \mathit{dm} \;||\; c_2 := \mathit{req}$ **END**
$\mathit{Reg}_{\mathcal{E},2} \;\hat{=}\;$ **SELECT** $c_2 = \mathit{ack}$ **THEN** $c_1 := \mathit{ack}$ **END**

The action schedules in such a machine $\mathcal{E}'$, corresponding to $\mathit{schedule}_{\mathcal{M}}$ and $\mathit{schedule}_{\mathcal{A}''}$ above, are then the following:

$\mathit{schedule}_{\mathcal{R}eg}[\mathit{Reg}_{\mathcal{E},1}, \mathit{Reg}_{\mathcal{E},2} \,/\, \mathit{Reg}_1, \mathit{Reg}_2] \;\hat{=}\; \mathit{Reg}_{\mathcal{E},1} \;;\; \mathit{Reg}_{\mathcal{E},2}$
$\mathit{schedule}_{\mathcal{E}^1} \;\hat{=}\; (E_{11} \;;\; E_{13}) \,[]\, E_{12}$

5.2. Introduction of communication channels

The most essential transformation in a synthesis step, described above, is the introduction of communication channels, which includes insertion of new variables modelling the channels and a subsequent set of atomicity refinements, i.e., decomposition of atomic actions involved into a number of separate atomic entities. Three main factors determine how a communication channel is created in practice. Firstly, the library component $\mathcal{M}$ in question can be either the master (active) or the slave (passive) with respect to the introduced channel. Secondly, communication on the channel may directly involve either one or two operations of the component $\mathcal{M}$, as explained in Section 3.2. Thirdly, from the atomicity refinement point of view, either one or two operations can be split by the procedure.

Let us now consider the introduction of a single channel $c(d)$ in detail. Here $c \in com$, initialised to *ack*, and d is a list of data variables of any type. Quite often only a communication variable c is introduced, without any new data variables d. Moreover, the type of the channel (push, pull, biput) affects how the variables d are actually accessed by the operations involved. Hence, when we mention in the following discussion substitutions or operations "accessing d", it is meant to be taken as an option, not as a requirement. The discussion below applies also to a more complex channel type $com_{m,n}$ with several request and acknowledgement states. In that case, new states or values are introduced for an existing communication variable c, rather than introducing a completely new variable at each step.

Observe that even though we discuss below creating component-specific operations $M_{\mathcal{A},1}$ and $M_{\mathcal{A},2}$, or simply $M_{\mathcal{A}}$ (cf. Section 5.1), by introducing a single communication channel $c(d)$, actually several such introduction steps are required if these operations take care of communication on several channels. Furthermore, some auxiliary application-dependent variable introductions and/or atomicity refinements might be needed as well in order to obtain the actual component-specific operations. However, the following refinement principles apply independently of whether a transformation step extracts the final component-specific actions or intermediate forms of some kind.

(**Case 1**). We first study the case where a single atomic action of a machine $\mathcal{A}1$ is decomposed to obtain the operation sequence $M_{\mathcal{A},1}$ **;** $M_{\mathcal{A},2}$ which has a corresponding equivalent in the library component $\mathcal{M}$. Assume that the pre-defined module $\mathcal{M}$ is the master of the introduced channel $c(d)$, and that the operations $M_{\mathcal{A},1}$ and $M_{\mathcal{A},2}$ are composed of communication events on the variable c and the substitutions $S_{M,1}$ and $S_{M,2}$, respectively, accessing the new data variables d. Let the initial system $\mathcal{A}1$ be of the form

MACHINE $\mathcal{A}1$
...
OPERATIONS
$A \;\widehat{=}\;$ **SELECT** P **THEN** S **END**;
...
SCHEDULING
$\mathcal{A}1 \widehat{=} A \ldots$
END

where A is the operation that we are going to split, P is a boolean expression, and S is a substitution from which the component-specific substitutions $S_{M,1}$ and $S_{M,2}$ can be extracted. By introducing the fresh channel variables c and d, and refining the substitution S into an atomic sequence of three substitutions accessing the variables d, we first obtain the following machine refinement $\mathcal{A}1'$:

REFINEMENT $\mathcal{A}1'$
REFINES $\mathcal{A}1$
...
VARIABLES c, d
...
OPERATIONS
$A' \;\hat{=}\;$ **SELECT** P **THEN** $S_{M,1}(d); c := req; S_A(d); c := ack; S_{M,2}(d)$ **END**;
...
SCHEDULING
$\mathcal{A}1' \hat{=} A' \ldots$
MAPPINGS
$A \leq A'$
...
END

Notice that the new variable c does not yet have any significant role in the system $\mathcal{A}1'$. Next, to complete the channel introduction procedure, the operation A' is split into three separate atomic actions using the communication variable c and sequential scheduling. This atomicity refinement yields the machine $\mathcal{A}1''$ below. The targeted component-specific operation sequence has then been created. It is a master-side communication sequence, in which c is first set to the request state *req* by the operation $M_{\mathcal{A},1}$, and then the acknowledgement state *ack*, set by the slave action A'', is detected by the subsequent operation $M_{\mathcal{A},2}$. The substitutions $S_{M,1}$, S_A and $S_{M,2}$ are executed exactly in the same order as in the previous machine $\mathcal{A}1'$, but in $\mathcal{A}1''$ this execution sequence is non-atomic. The transformation is correct provided that we can show the correctness of refining A' into $(M_{\mathcal{A},1}\ ;\ M_{\mathcal{A},2})\ []\ A''$ as required by the proof obligation (4). For this, we need to find a suitable invariant R which is application dependent.

REFINEMENT $\mathcal{A}1''$
REFINES $\mathcal{A}1'$
...
OPERATIONS
$M_{\mathcal{A},1} \;\hat{=}\;$ **SELECT** P **THEN** $S_{M,1}(d) \;||\; c := req$ **END**;
$A'' \;\hat{=}\;$ **SELECT** $c = req$ **THEN** $S_A(d) \;||\; c := ack$ **END**;
$M_{\mathcal{A},2} \;\hat{=}\;$ **SELECT** $c = ack$ **THEN** $S_{M,2}(d)$ **END**;
...
SCHEDULING
$\mathcal{A}1'' \hat{=} (M_{\mathcal{A},1}\ ;\ M_{\mathcal{A},2})\ []\ A'' \ldots$
MAPPINGS
$A' \leq (M_{\mathcal{A},1}\ []\ M_{\mathcal{A},2}\ []\ A'')^*$,
...
END

It is interesting to observe that the case where only one operation of the component $\mathcal{M}$ is involved in communication on the channel $c(d)$ can actually be viewed as a special case of the refinement presented. We namely have that if $S_{M,2} \mathrel{\hat{=}} \mathit{skip}$, we can replace the operation sequence $M_{\mathcal{A},1}$ **;** $M_{\mathcal{A},2}$ with the single action $M_{\mathcal{A}}$ given as

$$M_{\mathcal{A}} \;\hat{=}\; \textbf{SELECT } c = \mathit{ack} \wedge P \textbf{ THEN } S_{M,1}(d) \parallel c := \mathit{req} \textbf{ END}$$

(**Case 2**). Let us still consider the above situation, where the master-type operation sequence $M_{\mathcal{A},1}$ **;** $M_{\mathcal{A},2}$ is stepwise extracted by introducing a communication channel $c(d) \in \mathit{com}$, but now assume that two sequentially scheduled actions A_1 and A_2 need to be decomposed. The initial system $\mathcal{A}2$ is then of the form

MACHINE $\mathcal{A}2$
...
OPERATIONS
$A_1 \;\hat{=}\;$ **SELECT** P_1 **THEN** S_1 **END**;
$A_2 \;\hat{=}\;$ **SELECT** P_2 **THEN** S_2 **END**;
...
SCHEDULING
$\mathcal{A}2 \mathrel{\hat{=}} (A_1 \textbf{ ; } A_2) \ldots$
END

As before, the first refinement step is to introduce the channel variables c and d and to split the substitutions S_1 and S_2 into atomic sequences where the component-specific substitutions $S_{M,1}$ and $S_{M,2}$ are present. Now both S_1 and S_2 are locally divided into two parts accessing the data variables d, and assignments to c are placed between these parts similarly to in the above. Hence, the following machine $\mathcal{A}2'$ is obtained:

REFINEMENT $\mathcal{A}2'$
REFINES $\mathcal{A}2$
...
VARIABLES c, d
...
OPERATIONS
$A'_1 \;\hat{=}\;$ **SELECT** P_1 **THEN** $S_{M,1}(d); c := \mathit{req}; S_{A,1}(d)$ **END**;
$A'_2 \;\hat{=}\;$ **SELECT** P_2 **THEN** $S_{A,2}(d); c := \mathit{ack}; S_{M,2}(d)$ **END**;
...
SCHEDULING
$\mathcal{A}2' \mathrel{\hat{=}} (A'_1 \textbf{ ; } A'_2) \ldots$
MAPPINGS
$A_1 \leq A'_1,\ A_2 \leq A'_2,\ \ldots$
END

The second step is the atomicity refinement, where each of the actions A'_1 and A'_2 is split into two separate operations using the communication variable c and sequential scheduling. This completes the channel introduction procedure. The resulting system is the machine refinement $\mathcal{A}2''$ given as

REFINEMENT $\mathcal{A}2''$
REFINES $\mathcal{A}2'$
...
OPERATIONS
$M_{\mathcal{A},1} \;\hat{=}\;$ **SELECT** P_1 **THEN** $S_{M,1}(d) \,||\, c := req$ **END**;
$A''_1 \;\hat{=}\;$ **SELECT** $c = req$ **THEN** $S_{A,1}(d)$ **END**;
$A''_2 \;\hat{=}\;$ **SELECT** P_2 **THEN** $S_{A,2}(d) \,||\, c := ack$ **END**;
$M_{\mathcal{A},2} \;\hat{=}\;$ **SELECT** $c = ack$ **THEN** $S_{M,2}(d)$ **END**;
...
SCHEDULING
$\mathcal{A}2'' \;\hat{=}\; (M_{\mathcal{A},1} \,;\, M_{\mathcal{A},2}) \,[]\, (A''_1 \,;\, A''_2) \ldots$
MAPPINGS
$A'_1 \leq (M_{\mathcal{A},1} \,[]\, A''_1)^*$,
$A'_2 \leq (A''_2 \,[]\, M_{\mathcal{A},2})^*$,
...
END

(**Case 3**). Now, consider the case where the module $\mathcal{M}$ is the slave of the introduced channel $c(d) \in com$. Then a component-specific operation detects the request state *req* on c and sets it to the acknowledgement state *ack* as a response. From the point of view of the above refinements (Case 1 and 2) this simply means that the roles of the operations are switched; otherwise the transformations are the same. Hence, when splitting a single action (Case 1), a single component-specific operation $M_{\mathcal{A}}$ is created. It corresponds to the slave action A'' of the above machine $\mathcal{A}1''$ and has the form

$$M_{\mathcal{A}} \;\hat{=}\; \textbf{SELECT } c = req \textbf{ THEN } S_M(d) \,||\, c := ack \textbf{ END}$$

The operation sequence $M_{\mathcal{A},1} \,;\, M_{\mathcal{A},2}$ in $\mathcal{A}1''$ becomes $A_1 \,;\, A_2$, where

$$A_1 \;\hat{=}\; \textbf{SELECT } P \textbf{ THEN } S_{A,1}(d) \,||\, c := req \textbf{ END}$$
$$A_2 \;\hat{=}\; \textbf{SELECT } c = ack \textbf{ THEN } S_{A,2}(d) \textbf{ END}$$

(**Case 4**). When the module $\mathcal{M}$ is the slave of the introduced channel $c(d) \in com$, and two actions A_1 and A_2 have to be split, the procedure is the same as in Case 2. However, the roles of the actions are switched similarly to in Case 3. Thus, the slave-type operation sequence $A''_1 \,;\, A''_2$ in $\mathcal{A}2''$ becomes the component-specific sequence $M_{\mathcal{A},1} \,;\, M_{\mathcal{A},2}$ with

$$M_{\mathcal{A},1} \;\hat{=}\; \textbf{SELECT } c = req \textbf{ THEN } S_{M,1}(d) \textbf{ END}$$
$$M_{\mathcal{A},2} \;\hat{=}\; \textbf{SELECT } P_2 \textbf{ THEN } S_{M,2}(d) \,||\, c := ack \textbf{ END}$$

and, correspondingly, the master-type sequence $M_{\mathcal{A},1} \,;\, M_{\mathcal{A},2}$ in $\mathcal{A}2''$ is turned into $A''_1 \,;\, A''_2$, where

$$A''_1 \;\hat{=}\; \textbf{SELECT } P_1 \textbf{ THEN } S_{A,1}(d) \,||\, c := req \textbf{ END}$$
$$A''_2 \;\hat{=}\; \textbf{SELECT } c = ack \textbf{ THEN } S_{A,2}(d) \textbf{ END}$$

6. Synthesis example

As an example of the synthesis process, consider the action system $\mathcal{E}$ below. We assume that $\mathcal{E}$ operates within an environment, modelled by another abstract machine,

which instantiates $\mathcal{E}$ in its *scheduling*-clause as the component $\mathcal{E}(l, din, r, dout)$, where $(l \in com) \wedge (r \in com) \wedge (dout \in data) \wedge (din \in data)$, and the communication variables l and r are both initialised to *ack*:

MACHINE $\mathcal{E}$
OPERATIONS
$l, r, dout \leftarrow E(l, din, r, dout) \;\hat{=}$
SELECT $l = req \wedge r = ack$
THEN ANY $dout'$ **WHERE** $F(dout', din, dout)$ **THEN** $dout := dout'$ **END**
$\| \; r := req \;\|\; l := ack$
END
END

The machine $\mathcal{E}$ is an abstract model of an asynchronous system. It has one passive input port $l(din)$ and one active output port $r(dout)$, and its behaviour is the following. First, the environment selects a value for the data input *din* and activates the machine $\mathcal{E}$ by setting the channel l to the request state *req*. Then $\mathcal{E}$ computes a new value for the data output *dout* using the relation F. Observe that F is not explicitly specified in this generic example. The machine $\mathcal{E}$ sends the data *dout* to the environment via the channel r by setting r to the request state *req*. Simultaneously, an acknowledgement *ack* is issued through the channel l. This indicates that the environment may send new data *din* to $\mathcal{E}$ via l, but computation in $\mathcal{E}$ is blocked until the environment has stored the value of *dout* and issued an acknowledgement through r.

Let us now assume that we have a library of pre-defined asynchronous components available and that we intend to implement the abstract system specification $\mathcal{E}$ stepwise as a composition of four components belonging to this library: register $\mathcal{R}eg$, function $\mathcal{F}unc$, release $\mathcal{R}$ and suspend $\mathcal{S}usp$. Below we discuss these components and related synthesis steps separately.

6.1. Register component

The predicate F in the operation E of the machine $\mathcal{E}$ refers also to the variable *dout* itself. In other words, the next value of *dout* depends on the current value of *dout*. Furthermore, new input data *din* can arrive from the environment before communication on the channel r has been completed. This indicates that a storage element, a register, is needed for the variable *dout*. Hence, the register component $\mathcal{R}eg$, given below, is extracted as the first synthesis step.

MACHINE $\mathcal{R}eg$
OPERATIONS
$b, dout \leftarrow Reg_1(a, din) \;\hat{=}\;$ **SELECT** $a = req$ **THEN** $dout := din \;\|\; b := req$ **END**;

$a \leftarrow Reg_2(b) \;\hat{=}\;$ **SELECT** $b = ack$ **THEN** $a := ack$ **END**
SCHEDULING
$\mathcal{R}eg(a, din, b, dout) \;=\; Reg_1 \;;\; Reg_2$
END

The synthesis step splits the operation E of $\mathcal{E}$ into five separate parts, two of which (Reg_1, Reg_2) belong to the register $\mathcal{R}eg$ and the others (E_{11}, E_{12}, E_{13}) to the refined machine

$\mathcal{E}^1$ given below. For this, we introduce two fresh communication variables $c_1, c_2 \in com$ and a new data variable $dm \in data$ which acts as the data input of the extracted register component, the variable *dout* being the data output. Hence, we create the channels $c_1(dm)$ and $c_2(dout)$ through which $\mathcal{E}^1$ and $\mathcal{R}eg$ communicate. Because $\mathcal{R}eg$ is the slave of $c_1(dm)$ the refinement Case 3 of Section 5.2 is applied to introduce the channel $c_1(dm)$, so the single action E of $\mathcal{E}$ is first split into three parts: E_{11}, E_{13} and

$$r, dout \leftarrow Reg_{\mathcal{E}} \;\hat{=}\; \textbf{SELECT } c_1 = req \textbf{ THEN } dout := dm \,||\, c_1 := ack \,||\, r := req \textbf{ END}$$

with the scheduling $(E_{11}\ \textbf{;}\ E_{13})$ [] $Reg_{\mathcal{E}}$. Then, as the component $\mathcal{R}eg$ is the master of the channel $c_2(dout)$, the refinement Case 1 of Section 5.2 is used to split the intermediate operation $Reg_{\mathcal{E}}$ further into three parts: E_{12} and

$$dout \leftarrow Reg_{\mathcal{E},1} \;\hat{=}\; \textbf{SELECT } c_1 = req \textbf{ THEN } dout := dm \,||\, c_2 := req \textbf{ END}$$
$$Reg_{\mathcal{E},2} \;\hat{=}\; \textbf{SELECT } c_2 = ack \textbf{ THEN } c_1 := ack \textbf{ END}$$

with the scheduling $(Reg_{\mathcal{E},1}\ \textbf{;}\ Reg_{\mathcal{E},2})$ [] E_{12}. The sequential scheduling of the two component-specific actions can now be replaced with the reference to the actual component $\mathcal{R}eg$. This yields the system $\mathcal{E}^1$ below, which is a refinement of the initial abstract machine $\mathcal{E}$, containing $\mathcal{R}eg$ as a component.

The machine $\mathcal{E}^1$ activates $\mathcal{R}eg$ by computing a new value for the data variable *dm* and setting the channel c_1 to the request state in the operation E_{11}. Then $\mathcal{R}eg$ copies the value of *dm* to the variable *dout* and performs a communication cycle on the channel c_2, which activates $\mathcal{E}^1$ to send *dout* to the environment via the channel r in the operation E_{12}. After the cycle on c_2, $\mathcal{R}eg$ sets c_1 to the acknowledgement state, and $\mathcal{E}^1$ executes finally the operation E_{13}, where the channel l is set to the acknowledgement state.

REFINEMENT $\mathcal{E}^1$
REFINES $\mathcal{E}$
EXTENDS $\mathcal{R}eg$
VARIABLES
c_1, c_2, dm
INVARIANT
$c_1 \in com \wedge c_2 \in com \wedge dm \in data$
INITIALISATION
$c_1 := ack \,||\, c_2 := ack \,||\, dm :\in data$
OPERATIONS
$E_{11}(l, din, dout) \;\hat{=}$
SELECT $l = req \wedge r = ack$
THEN **ANY** dm' **WHERE** $F(dm', din, dout)$ **THEN** $dm := dm'$ **END**
$\,||\, c_1 := req$
END;

$r \leftarrow E_{12} \;\hat{=}\; \textbf{SELECT } c_2 = req \textbf{ THEN } c_2 := ack \,||\, r := req \textbf{ END}$;
$l \leftarrow E_{13} \;\hat{=}\; \textbf{SELECT } c_1 = ack \textbf{ THEN } l := ack \textbf{ END}$

SCHEDULING
$\mathcal{E}^1(l, din, r, dout) \;\hat{=}\; (E_{11}\ ;\ E_{13})\ []\ E_{12}\ []\ \mathcal{R}eg(c_1, dm, c_2, dout)$
MAPPINGS
$E \leq (E_{11}\ []\ E_{13}\ []\ E_{12}\ []\ \mathcal{R}eg(c_1, dm, c_2, dout))*$
END

6.2. Function component

The second synthesis step places the computation of the data variable *dm*, the input of $\mathcal{R}eg$, to the dedicated component machine $\mathcal{F}unc$ defined by

MACHINE $\mathcal{F}unc$
OPERATIONS
$b, dout \leftarrow Func_1(a, din_1, din_2) \;\hat{=}$
SELECT $a = req$
THEN **ANY** $dout'$ **WHERE** $F(dout', din_1, din_2)$ **THEN** $dout := dout'$ **END**
$||\ b := req$
END;

$a \leftarrow Func_2(b) \;\hat{=}\;$ **SELECT** $b = ack$ **THEN** $a := ack$ **END**
SCHEDULING
$\mathcal{F}unc(a, din_1, din_2, b, dout, F) \;=\; Func_1\ ;\ Func_2$
END

This library component has the passive input port $a(din_1, din_2) \in com$ and the active output port $b(dout) \in com$. Notice that also the predicate F is viewed as a parameter in the *scheduling*-clause. In the synthesis procedure, the formal ports $a(din_1, din_2)$ and $b(dout)$ are replaced with the actual push channels $c_3(din, dout)$ and $c_1(dm)$, respectively, where $c_3 \in com$ is the new communication variable introduced in the transformation step. The resulting machine $\mathcal{E}^2$, which is a refinement of $\mathcal{E}^1$, is given below.

The component $\mathcal{F}unc$ is the slave of the channel $c_3(din, dout)$. In order to extract $\mathcal{F}unc$, the operations E_{11} and E_{13} of the system $\mathcal{E}^1$ are split into two parts each by applying the refinement Case 4 of Section 5.2. The action E_{11} is transformed into E_{21} and $Func_{\mathcal{E},1}$, and the action E_{13} into E_{23} and $Func_{\mathcal{E},2}$, where the component-specific operations are defined as

$Func_{\mathcal{E},1}(din, dout) \;\hat{=}$
SELECT $c_3 = req$
THEN **ANY** dm' **WHERE** $F(dm', din, dout)$ **THEN** $dm := dm'$ **END** $||\ c_1 := req$
END
$Func_{\mathcal{E},2} \;\hat{=}\;$ **SELECT** $c_1 = ack$ **THEN** $c_3 := ack$ **END**

and the scheduling is $(E_{21}\ ;\ E_{23})\ []\ (Func_{\mathcal{E},1}\ ;\ Func_{\mathcal{E},2})$. The sequential scheduling of the component-specific actions is again replaced with the reference to the actual component $\mathcal{F}unc$, giving the system $\mathcal{E}^2$ below.

The system $\mathcal{E}^2$ activates $\mathcal{F}unc$ by executing the operation E_{21}, where the current values of the variables *dout* and *din* are sent to $\mathcal{F}unc$ via the new channel c_3. The function component then carries out the data assignment

ANY dm' **WHERE**
$F(dm', din, dout)$
THEN
$dm := dm'$
END

and sends the result data *dm* to the register component $\mathcal{R}eg$ via the channel c_1 which was created in the first decomposition step.

REFINEMENT $\mathcal{E}^2$
REFINES $\mathcal{E}^1$
EXTENDS $\mathcal{R}eg, \mathcal{F}unc$
VARIABLES
c_1, c_2, c_3, dm
INVARIANT
$c_1 \in com \wedge c_2 \in com \wedge dm \in data \wedge c_3 \in com$
DEFINITIONS
$F(\cdots) == \ldots$
INITIALISATION
$c_1 := ack \;||\; c_2 := ack \;||\; c_3 := ack \;||\; dm :\in data$
OPERATIONS
$E_{21}(l, r) \;\hat{=}\;$ **SELECT** $l = req \wedge r = ack$ **THEN** $c_3 := req$ **END**;

$r \leftarrow E_{22} \;\hat{=}\;$ **SELECT** $c_2 = req$ **THEN** $c_2 := ack \;||\; r := req$ **END**;

$l \leftarrow E_{23} \;\hat{=}\;$ **SELECT** $c_3 = ack$ **THEN** $l := ack$ **END**
SCHEDULING
$\mathcal{E}^2(l, din, r, dout) \;\hat{=}\; (E_{21} \,;\, E_{23}) \,[]\, E_{22} \,[]\, \mathcal{R}eg(c_1, dm, c_2, dout) \,[]\, \mathcal{F}unc(c_3, din, dout, c_1, dm, F)$
MAPPINGS
$E_{11} \leq (E_{21} \,[]\, \mathcal{F}unc(c_3, din, dout, c_1, dm, F)).Func_1)^*$
$E_{12} \leq E_{22}$
$E_{13} \leq (E_{23} \,[]\, \mathcal{F}unc(c_3, din, dout, c_1, dm, F)).Func_2)^*$
END

6.3. Final synthesis step

We complete our example by extracting two distinct library components based on the three operations of $\mathcal{E}^2$. The component machines are called $\mathcal{R}$ (release) and $\mathcal{S}usp$ (suspend), defined as follows:

MACHINE $\mathcal{R}$
OPERATIONS
$a, b, bsy \leftarrow R_1(a) \;\hat{=}\;$ **SELECT** $a = req$ **THEN** $b := req \;||\; a := ack \;||\; bsy := true$ **END**;

$bsy \leftarrow R_2(b) \;\hat{=}\;$ **SELECT** $b = ack$ **THEN** $bsy := false$ **END**
SCHEDULING
$\mathcal{R}(a, b, bsy) \;=\; R_1 \,;\, R_2$
END

MACHINE $\mathcal{S}usp$
OPERATIONS
$b \leftarrow Susp_1(a, bsy) \;\hat{=}\;$ **SELECT** $a = req \wedge \neg bsy$ **THEN** $b := req$ **END**;

$a \leftarrow Susp_2(b) \;\hat{=}\;$ **SELECT** $b = ack$ **THEN** $a := ack$ **END**
SCHEDULING
$\mathcal{S}usp(a, b, bsy) = Susp_1$ **;** $Susp_2$
END

They both have two communication ports: the passive port a and the active port b. Furthermore, $\mathcal{R}$ outputs the boolean signal *bsy* setting it to *true* whenever a communication cycle on the active port b begins, and back to *false* when a cycle on b ends. The component $\mathcal{S}usp$, in turn, reads the signal *bsy*, allowing a new communication cycle on its active port b to start only when $bsy = false$. In the system derivation, the formal parameters a and b of $\mathcal{R}$ are replaced with the actual interface variables c_2 and r, respectively. In the case of $\mathcal{S}usp$, a and b are replaced with l and c_3, respectively. The parameter *bsy* becomes a variable of the same name, shared by the components $\mathcal{R}$ and $\mathcal{S}usp$. The final system $\mathcal{E}^3$ is then a refinement of $\mathcal{E}^2$ and is given below. Observe that $\mathcal{E}^3$ does not have operations of its own, but the functionality comes completely from the four component machines.

REFINEMENT $\mathcal{E}^3$
REFINES $\mathcal{E}^2$
EXTENDS $\mathcal{R}eg, \mathcal{F}unc, \mathcal{R}, \mathcal{S}usp$
VARIABLES
c_1, c_2, c_3, bsy, dm
INVARIANT
$c_1 \in com \wedge c_2 \in com \wedge c_3 \in com \wedge bsy \in BOOL \wedge dm \in data$
DEFINITIONS
$F(\cdots) == \ldots$
INITIALISATION
$c_1 := ack \;||\; c_2 := ack \;||\; c_3 := ack \;||\; bsy := false \;||\; dm :\in data$
SCHEDULING
$\mathcal{E}^3(l, din, r, dout) \;\hat{=}\; \mathcal{R}eg(c_1, dm, c_2, dout)$
$[]\; \mathcal{F}unc(c_3, din, dout, c_1, dm, F)$
$[]\; \mathcal{R}(c_2, r, bsy)\; []\; \mathcal{S}usp(l, c_3, bsy)$
MAPPINGS
$E_{21} \leq (\mathcal{S}usp(l, c_3, bsy).Susp_1\; []\; \mathcal{R}(c_2, r, bsy).R_2)^*$
$E_{22} \leq \mathcal{R}(c_2, r, bsy).R_1$
$E_{23} \leq \mathcal{S}usp(l, c_3, bsy).Susp_2$
END

In this transformation, we do not insert new communication variables of the type *com*, which means that the refinements discussed in Section 5.2 do not apply here. Instead, a boolean variable *bsy* ("busy") is introduced in order to move the detection of the condition $r = ack$ to a different location, making extraction of $\mathcal{R}$ and $\mathcal{S}usp$ possible. The action E_{21} of $\mathcal{E}^2$ is split into $Susp_{\mathcal{E},1}$ and $R_{\mathcal{E},2}$, and the actions E_{22} and E_{23} of $\mathcal{E}^2$ are implemented by $R_{\mathcal{E},1}$ and $Susp_{\mathcal{E},2}$, respectively. The component-specific actions mentioned are defined by

$Susp_{\mathcal{E},1}(l) \;\hat{=}\;$ **SELECT** $l = req \wedge \neg bsy$ **THEN** $c_3 := req$ **END**
$l \leftarrow Susp_{\mathcal{E},2} \;\hat{=}\;$ **SELECT** $c_3 = ack$ **THEN** $l := ack$ **END**

$r \leftarrow R_{\mathcal{E},1} \;\hat{=}\;$ **SELECT** $c_2 = req$ **THEN** $r := req \,||\, c_2 := ack \,||\, bsy := true$ **END**
$R_{\mathcal{E},2}(r) \;\hat{=}\;$ **SELECT** $r = ack$ **THEN** $bsy := false$ **END**

Their scheduling is: $(Susp_{\mathcal{E},1}$ **;** $Susp_{\mathcal{E},2})$ [] $(R_{\mathcal{E},1}$ **;** $R_{\mathcal{E},2})$. The final machine $\mathcal{E}^3$ is then obtained by replacing these component-specific operation sequences with the references to the actual library components $\mathcal{S}usp$ and $\mathcal{R}$.

The component $\mathcal{R}$ is enabled by the register component $\mathcal{R}eg$ via the channel c_2. It initiates a communication cycle with the environment on the channel r and immediately sends an acknowledgement back to $\mathcal{R}eg$ setting the introduced boolean variable *bsy* to *true*. Hence, $\mathcal{R}$ releases the communication cycles on the channels c_1, c_2, c_3 and l to be completed while output data *dout* is transferred from the system $\mathcal{E}^3$ to the environment through r. The component $\mathcal{S}usp$ reads the variable *bsy* and suspends computation on new input data *din*, sent by the environment via the channel l, until the component $\mathcal{R}$ has received an acknowledgement on r and set the control signal *bsy* back to *false*.

7. Conclusions

We have proposed an approach to component-based asynchronous system design within the B Method and its supporting tools. The main idea is to extract system components in a stepwise manner from the initial specification. At each extraction step new asynchronous communication channels are introduced. We show how to give a model for components. The model encapsulates the communication protocols needed to interact with the component. The interface model additionally gives types for the interface variables. The compatibility of each component is ensured via the refinement proofs and decomposition approach. The methodology proposed here also gives insight into how to design components.

There has been some work on defining component interfaces in the context of formal component-based design approaches; see e.g. [11] for a discussion on interface models and [14] for extending the type system concepts to capture component interaction. However, these seldom come with a practical methodology and tool support, which is our main focus here. The tool support of the B Method helps in carrying out proofs in connection to component compatibility and typing for interface variables. Moreover, the entire derivation task and the component libraries can be administered within the tool.

We used the B Method almost as such with minor extensions mainly needed to specify alternative scheduling strategies for actions in a *scheduling*-clause and explicitly giving refinement relations in the *mappings*-clause. The first clause is mainly syntactic sugaring as every scheduling could be coded in the actions of the *operations*-clause. The second clause is a real extension, but also it can be easily supported by the tools. The *scheduling*-clause is inspired by the work of Butler [8] who studies a more general process concept for the B Method to tie it together with the CSP approach [13].

The *Refinement Calculator* [25] is a tool that supports development of correct programs within the refinement calculus framework. It is a tool for program refinement based on the HOL theorem prover. Until now, the tool has not had proper support for action systems. However, a method for proving the correctness of the implementations of asynchronous modules has been mechanised within the HOL theorem prover [21] which supports more

general verification techniques than the tools studied in this paper. In the future we can envisage a situation where the verifications of the low-level implementations are carried out within HOL and the high-level design and component libraries are supported by the tools of the B Method.

Acknowledgement

The third author was financed by the Academy of Finland, Project No 780.

Appendix A. Proving the correctness of refinement

Let us consider the abstract machine $\mathcal{A}$ below and its refinement $\mathcal{C}$ as described in Section 4:

MACHINE $\mathcal{A}$
VARIABLES
x
INVARIANT
$I(x)$
INITIALISATION
$x := x_0$
OPERATIONS
$A_1 \;\hat{=}\;$ **SELECT** P_1 **THEN** S_1 **END**;
...
$A_m \;\hat{=}\;$ **SELECT** P_m **THEN** S_m **END**
SCHEDULING
$\mathcal{A} \hat{=} \mathit{schedule}_{\mathcal{A}}(A_1, \ldots, A_m)$
END

REFINEMENT $\mathcal{C}$
REFINES
$\mathcal{A}$
VARIABLES
y
INVARIANT
$R(x, y)$
INITIALISATION
$y := y_0$
OPERATIONS
$C_1 \;\hat{=}\;$ **SELECT** Q_1 **THEN** T_1 **END**;
...
$C_n \;\hat{=}\;$ **SELECT** Q_n **THEN** T_n **END**
SCHEDULING
$\mathcal{C} \hat{=} \mathit{schedule}_{\mathcal{C}}(C_1, \ldots, C_n)$
MAPPINGS
...
END

In order to prove that the refinement $\mathcal{C}$ on the variables y is a refinement of the action system $\mathcal{A}$ on the variables x using the invariant $R(x, y)$, a number of proof obligations must be satisfied [1]. Here we elaborate on these proof obligations.

First, the invariant R of the refinement should not contradict the invariant I of the abstract machine.

$$(\exists(x, y).\ I \wedge R) \tag{1}$$

Furthermore, the initialisation $y := y_0$ in $\mathcal{C}$ establishes a situation where the initialisation $x := x_0$ in $\mathcal{A}$ cannot fail to establish the invariant R:

$$\mathrm{wp}(y := y_0, \neg \mathrm{wp}(x := x_0, \neg R)). \tag{2}$$

Moreover, we need to prove that the actions in $\mathcal{A}$ are data refined by the actions in $\mathcal{C}$. In the case where an action A_i in $\mathcal{A}$ is data refined by one action C_j in $\mathcal{C}$ under invariant R, we have an entry $A_i \leq C_j$ in the *mappings*-clause:

MAPPINGS
$A_i \leq C_j$,
...

We should then prove that there is an action C_j in $\mathcal{C}$ such that C_j establishes a situation where action A_i in $\mathcal{A}$ cannot fail to maintain R:

$$(\forall(x, y).\ I \wedge R \Rightarrow \mathrm{wp}(C_i, \neg \mathrm{wp}(A_i, \neg R))) \tag{3}$$

where $1 \leq i \leq m$.

The proof obligations (1)–(3) above can be generated automatically and checked using the theorem-proving environments associated with the B Method [1,17]. See Waldén and Sere [26] for further details on refining action systems within B.

In the case where we introduce new actions in $\mathcal{C}$ ($n > m$), we have the case that an action in $\mathcal{A}$ is refined by a composition of actions in $\mathcal{C}$:

MAPPINGS
$A_i \leq (C_j \ [] \ \ldots \ [] \ C_l)^*$,
...

where $1 \leq i \leq m$ and $1 \leq j < l \leq n$. The entry states that the action A_i in $\mathcal{A}$ is refined by the composition of the actions $C_j, \ldots, C_l$ in $\mathcal{C}$. For proving the correctness of this refinement we need to take into consideration the actual scheduling of the actions $C_j, \ldots, C_l$ from the *scheduling*-clause. This is done in such a way that all the compositions in the *scheduling*-clause of $\mathcal{C}$ are interpreted as non-deterministic composition using program counters as explained in Section 2.1, i.e., $(A\ ;\ B)$ is interpreted as $(A^{pc}\ []\ B^{pc})$. We can then interpret the scheduling in $\mathcal{C}$ as a non-deterministic choice of the actions, where program counters are given explicitly, $C_1^{pc}\ []\ \ldots\ []\ C_n^{pc}$. Hence, in the proofs we actually consider the program counters of the actions of $\mathcal{C}$ in the *mappings*-clause as well, $(C_j^{pc}\ []\ \ldots\ []\ C_l^{pc})^*$. When considering the program counters in this way, we make sure that there will be no interference from the other actions during the execution of the actions $C_j, \ldots, C_l$. We can give the refined, composed action in B notation as

WHILE $Q_j^{pc} \vee \ldots \vee Q_l^{pc}$ **DO** (**CHOICE** C_j^{pc} **OR** ... **OR** C_l^{pc} **END**) **END**

and call it D_i. The disjunction of the guards of the actions $C_j^{pc}, \ldots, C_l^{pc}$ form the guard of the loop.

The proof obligation for refining an action with composed actions can then be given as

$$(\forall(x, y).\ I \wedge R \Rightarrow \mathrm{wp}(D_i, \neg \mathrm{wp}(A_i, \neg R))), \tag{4}$$

for each entry $A_i \leq (C_j\ []\ \ldots\ []\ C_l)^*$ in the *mappings*-clause. Hence, for each action A_i in $\mathcal{A}$ the composed actions D_i of $\mathcal{C}$ should establish such a situation that A_i cannot fail to

maintain R. Condition (4) requires that the composed actions D_i terminate when executed in isolation,

$$(\forall(x, y).\ I \wedge R \Rightarrow \mathrm{wp}(D_i, \mathit{true})). \tag{5}$$

This condition is established when each action disables itself as is the case for every action in this paper.

The proof obligations generated for the weakest precondition of a loop to establish post-condition Q,

$$\mathrm{wp}(\textbf{WHILE}\ G\ \textbf{DO}\ S\ \textbf{END}, Q),$$

with the invariant R and the variant V are the following:

(i) $R \Rightarrow E \in \mathrm{NAT}$,
(ii) $(\forall(z).\ R \wedge G \Rightarrow \mathrm{wp}(n := V, \mathrm{wp}(S, n > V)))$,
(iii) $(\forall(z).\ R \wedge G \Rightarrow \mathrm{wp}(S, R))$,
(iv) $R \wedge \neg G \Rightarrow Q$,

where z denotes the variables modified within the loop. Thus, (i) the variant should be an expression yielding a natural number, and when the guard G of the loop holds, the body S should (ii) decrease the variant V and (iii) preserve the invariant. When the guard does not hold, (iv) the post-condition should be established.

Example. Studying the examples given in Sections 2 and 3 we can note that the action system $\mathcal{E}^1$, which has $\mathcal{R}eg$ as a component, is actually a refinement of the action system $\mathcal{E}$. Since $\mathcal{E}$ has fewer actions than $\mathcal{E}^1$, the refinement of the actions is given in the *mappings*-clause as follows:

REFINEMENT $\mathcal{E}^1$
REFINES $\mathcal{E}$
EXTENDS
$\mathcal{R}eg$
...
SCHEDULING
$\mathcal{E}^1(l, din, r, dout) \ \hat{=}\ (E_{11}\ ;\ E_{13})\ [\!]\ E_{12}\ [\!]\ \mathcal{R}eg(c_1, dm, c_2, dout)$
MAPPINGS
$E \leq (E_{11}\ [\!]\ E_{12}\ [\!]\ E_{13}\ [\!]\ \mathcal{R}eg(c_1, c_2, dm, dout))^*$
END

The tool is then used to generate the proof obligations (1), (2) and (4) for proving that $\mathcal{E}^1$ is a correct refinement of $\mathcal{E}$. Since we only have one entry in $\mathcal{E}^1$ containing composition of actions, we need not generate any proof obligation of type (3).

Firstly, the invariant of $\mathcal{E}^1$ should not contradict that of $\mathcal{E}$:

$$(1) \qquad (\exists(c_1, c_2, dm).\ \mathit{true} \wedge (c_1 \in \mathit{com} \wedge c_2 \in \mathit{com} \wedge dm \in \mathit{data})).$$

This is obviously true, since the invariant of $\mathcal{E}$ has the value *true*.

We should also prove that the initialisation in $\mathcal{E}^1$ is a refinement of the initialisation in $\mathcal{E}$ under the invariant of $\mathcal{E}^1$:

$$(2) \qquad \mathrm{wp}(\ c_1 := ack \parallel c_2 := ack \parallel dm :\in data,\ \neg\mathrm{wp}(skip, \neg(c_1 \in com \wedge c_2 \in com \wedge dm \in data))).$$

This can be simplified to the expression

$$ack \in com \wedge ack \in com \wedge \forall d.(d \in data \Rightarrow d \in data)$$

which is trivially *true*, since *com* is a set of the elements *ack* and *req*.

Moreover, we need to prove for the entry

$$E \leq (E_{11} \,[]\, E_{12} \,[]\, E_{13} \,[]\, \mathcal{R}eg(c_1, c_2, dm, dout))^*$$

in $\mathcal{E}^1$ that the action E is refined by the given composition of actions of $\mathcal{E}^1$ under the invariant of $\mathcal{E}^1$. We use $g\mathcal{E}^1$ to denote the disjunction of the guards of the actions in $\mathcal{E}^1$ and $\mathcal{R}eg$, where program counters are taken into account considering the scheduling of the actions. Furthermore, we denote the composition of actions $(E_{11}^{pc} \,[]\, E_{12}^{pc} \,[]\, E_{13}^{pc} \,[]\, Reg_1^{pc} \,[]\, Reg_2^{pc})$ as $s\mathcal{E}^1$. We then create the proof obligation

$$(4) \qquad (\forall(c_1, c_2, dm).\ true \wedge (c_1 \in com \wedge c_2 \in com \wedge dm \in data) \Rightarrow \mathrm{wp}((\textbf{WHILE}\ g\mathcal{E}^1\ \textbf{DO}\ s\mathcal{E}^1\ \textbf{END}), \neg\mathrm{wp}(E, \neg(c_1 \in com \wedge c_2 \in com \wedge dm \in data)))).$$

Since action E in $\mathcal{E}$ does not assign the variables c_1, c_2 and *dm*, the proof obligation can be simplified to

$$(\forall(c_1, c_2, dm).\ (c_1 \in com \wedge c_2 \in com \wedge dm \in data) \Rightarrow \mathrm{wp}((\textbf{WHILE}\ g\mathcal{E}^1\ \textbf{DO}\ s\mathcal{E}^1\ \textbf{END}), (c_1 \in com \wedge c_2 \in com \wedge dm \in data))),$$

which is trivially true since the variables c_1, c_2 and *dm* are assigned values of correct types in $\mathcal{E}^1$.

Finally, we have to show that the actions in $\mathcal{E}^1$ terminate.

$$(5) \qquad (\forall(c_1, c_2, dm).\ true \wedge (c_1 \in com \wedge c_2 \in com \wedge dm \in data) \Rightarrow \mathrm{wp}((\textbf{WHILE}\ g\mathcal{E}^1\ \textbf{DO}\ s\mathcal{E}^1\ \textbf{END}), true)).$$

This is true, since all the actions in $\mathcal{E}^1$ and $\mathcal{R}eg$ disable themselves. We can note that the actions E_{11}^{pc}, E_{13}^{pc}, Reg_1^{pc} and Reg_2^{pc} disable themselves merely due to the program counters considered in these actions.

Proof obligations (1) and (2) above can be generated and automatically or interactively proved by the current provers of the B tool. Hence, the tool assists us in proving that $\mathcal{E}^1$ is a correct refinement of $\mathcal{E}$. Taking the *scheduling*- and *mappings*-clauses into account, proof obligation (4) could also be automatically generated.

References

[1] J.-R. Abrial, The B-Book, Cambridge University Press, Cambridge, Great Britain, 1996.
[2] R.J.R. Back, Atomicity refinement in a refinement calculus framework, Reports on Computer Science and Mathematics, Technical Report A-141, Åbo Akademi University, Turku, Finland, 1993.
[3] R.J.R. Back, R. Kurki-Suonio, Decentralization of process nets with centralized control, in: Proc. of the 2nd ACM SIGACT–SIGOPS Symp. on Principles of Distributed Computing, 1983, pp. 131–142.

[4] R.J.R. Back och, K. Sere, From action systems to modular systems, in: Proc. of FME'94: Industrial Benefit of Formal Methods, Barcelona, Spain, October, 1994, LNCS, 873, Springer-Verlag, 1994, pp. 1–25.
[5] R.J.R. Back, K. Sere, Stepwise refinement of action systems, Structured Programming 12 (1991) 17–30.
[6] R.J.R. Back, K. Sere, Superposition refinement of reactive systems, Formal Aspects of Computing 8 (3) (1996) 324–346.
[7] M.M. Bonsangue, J.N. Kok, K. Sere, Developing object-based distributed system, in: Formal Methods for Open Object-based Distributed Systems FMOODS'99, Florence, Italy, February, 1999, Kluwer Academic Publishers, 1999.
[8] M.J. Butler, csp2B: A practical approach to combining CSP and B, in: J. Wing, J. Woodcock, J. Davies (Eds.), Proc. of FM'99—Formal Methods, Toulouse, France, September, 1999, LNCS, 1708, Springer-Verlag, 1999, pp. 490–508.
[9] M.J. Butler, M. Waldén, Distributed system development in B, in: H. Habrias (Ed.), Proc. of the First Conference on the B Method, Nantes, France, November, 1996, IRIN, 1996, pp. 155–168.
[10] ClearSy, Event B Reference Manual, v1, 2001.
[11] L. de Alfaro, T.A. Henzinger, Interface theories for component-based design, in: Proc. of the 1st International Workshop on Embedded Software, Springer-Verlag, 2001.
[12] E.W. Dijkstra, A Discipline of Programming, Prentice-Hall International, Englewood Cliffs, NJ, 1976.
[13] C.A.R. Hoare, Communicating Sequential Processes, Series in Computer Science, Prentice-Hall Int, 1985.
[14] E.A. Lee, Y. Xiong, System-level types for component-based design, in: First Workshop on Embedded Software EMSOFT2001, Lake Tahoe, CA, USA, 2001.
[15] R.J. Lipton, Reduction: A method of proving properties of parallel programs, Communications of the ACM 18 (12) (1975) 717–721.
[16] Stéria Méditerranée, Atelier B, France, 1996.
[17] D.S. Neilson, I.H. Sorensen, The B-Technologies: A system for computer aided programming, Including the B-Toolkit User's Manual, Release 3.2, B-Core (UK) Ltd., Oxford, UK, 1996.
[18] A. Peeters, Single-Rail Handshake Circuits, Ph.D. Thesis, Eindhoven University of Technology, The Netherlands, 1996.
[19] J. Plosila, Self-Timed Circuit Design—The Action Systems Approach, Ph.D. Thesis, University of Turku, Turku, Finland, 1999.
[20] J. Plosila, K. Sere, M. Waldén, Design with asynchronously communicating components, in: F. de Boer, M. Bonsangue, S. Graf, W.-P. de Roever (Eds.), Proc. of FMCO 2002 International Symposium on Formal Methods for Components and Objects, November, 2003, LNCS, Springer-Verlag, 2003.
[21] R. Ruksenas, Formal Development of Concurrent Components, Ph.D. Thesis, Turku Centre for Computer Science (TUCS), Turku, Finland, 2004.
[22] T. Seceleanu, Systematic Design of Synchronous Digital Circuits, Ph.D. Thesis, Turku Centre for Computer Science (TUCS), Turku, Finland, 2001.
[23] E. Sekerinski, K. Sere (Eds.), Program Development by Refinement, FACIT, Springer-Verlag, 1998.
[24] K. Sere, M. Waldén, Data refinement of remote procedures, Formal Aspects of Computing 12 (4) (2000) 278–297.
[25] J. von Wright, Program refinement by theorem prover, in: Proc. of Sixth BCS-FACS Refinement Workshop, January, 1994.
[26] M. Waldén, K. Sere, Reasoning about action systems using the B-Method, Formal Methods in System Design 13 (1) (1998) 5–35. Kluwer Academic Publishers.